Balanced Binary Tree

judge whether the tree is balanced binary tree

a binary tree in which the left and right subtrees of *every* node differ in height by no more than 1.

example 1 True

3

/ \

9 20

/ \

15 7

exaple two - False

1

/ \

2 2

/ \

3 3

/ \

4 4

Input : None, root(one node)

MethodL: recursive method with the hepler()

1. helper() find the height of the tree
2. differ between left and right subtrees is no more than 1 and left and right subtrees are both balanced -> BBT
3. return False

class Solution:

def balancedBT(self, root):

#corner case

if not root:

return True

if not root.left and not root.right:

return True

def helper(root):

if not root:

return 0

if not root.left and root.right:

return 1

left, right = helper(root.left), helper(root.right)

return max(left, right) + 1

left, right = helper(root.left), helper(root.right)

if abs(left - right) <= 1 and self.balanced(root.left) and self.balanced(root.right):

return True

return False

3

/ \

9 20

/ \

15 7

height left: 1 right: (right.left: 1, right.right: 1) 2 ->diff = 1

left subtree: balanced

right subtree: height left: 1, right: 1 -> diff = 0

left subtree: balanced, right: balance -> True -> True

1

/ \

2 2

/ \

3 3

/ \

4 4

[1] [2,2] cnt = 0 [3,3] cnt += 1 [4, 4] … cnt += 1

cnt >= 2: False

time complexity O(logn)

space complexity O(1)

class Solution:

def balancedBT(self, root):

#corner case

if not root:

return True

if not root.left and not root.right:

return True

#bfs

d = collections.deque()

d.append(root)

cnt = -1

while d:

for i in range(len(d)):

node = d.popleft()

if not node.left and not node.right:

if cnt == -1:

cnt = 0

if node.left:

d.append(node.left)

if node.right:

dappend(node.right)

if cnt == -1:

continue

else:

cnt += 1

return False

return True

1

/ \

2 2

/ \

3 3

/ \

4 4

d = [1] cnt = -1

d = [2,2] cnt = -1

d = [3, 3] cnt = 0, 1

d = [4, 4] cnt = 2

d = [] cnt = 3 return False